Automatic Inference of Structural Changes for Matching Across Program Versions

Miryung Kim, David Notkin, Dan Grossman
Computer Science & Engineering
University of Washington
Seattle WA, USA
{miryung, notkin, djg}@cs.washington.edu

Abstract

Mapping code elements in one version of a program to corresponding code elements in another version is a fundamental building block for many software engineering tools. Existing tools that match code elements or identify structural changes—refactorings and API changes—between two versions of a program have two limitations that we overcome. First, existing tools cannot easily disambiguate among many potential matches or refactoring candidates. Second, it is difficult to use these tools' results for various software engineering tasks due to an unstructured representation of results. To overcome these limitations, our approach represents structural changes as a set of high-level change rules, automatically infers likely change rules and determines method-level matches based on the rules. By applying our tool to several open source projects, we show that our tool identifies matches that are difficult to find using other approaches and produces more concise results than other approaches. Our representation can serve as a better basis for other software engineering tools.

1. Introduction

Matching code elements between two versions of a program is the underlying basis for various software engineering tools. For example, version merging tools identify possible conflicts among parallel updates by analyzing matched code elements [27], regression testing tools prioritize or select test cases that need to be re-run by analyzing matched code elements [15, 29, 30], and profile propagation tools use matching to transfer execution information between versions [32]. In addition, emerging interest in mining software repositories [1, 4] —studying program evolution by analyzing existing software project artifacts—is demanding more effective and systematic matching techniques. Our recent survey [23] found that existing techniques match code at particular levels (e.g., packages, classes, methods, or fields) based on closeness of names, structures, etc. Though intuitive, this general approach has some limitations. First, existing tools do not consider which set of structural changes are more likely to have happened; thus they cannot easily disambiguate among many potential matches or refactoring candidates. Second, existing tools represent the results as an unstructured, usually lengthy, list of matches or refactorings. Although this unstructured representation is adequate for conventional uses (e.g., transferring code-coverage information in profile-propagation tools), it may prevent existing tools from being broadly used in mining software repository research, which often demands an in-depth understanding of software evolution. It may also be an obstacle to software tools that could benefit from additional knowledge of the changes between versions.

Consider an example where a programmer reorganizes a chart drawing program by the type of a rendered object, moving axis-drawing classes from the package `chart` to the package `chart.axis`. Then, to allow toggling of tool tips by the user, she appends a `boolean` parameter to a set of chart-creation interfaces. Even though the goals of these transformations can be stated concisely in natural language, a method-level matching tool would report a list of matches that enumerates each method that has been moved and each interface that has been modified, and a refactoring reconstruction tool would report a list of low-level refactorings (see Table 1). One may have to examine hundreds or thousands of matches or refactorings before discovering that a few simple high-level changes took place. Moreover, if the programmer neglected to move one axis drawing class, this likely design error would be hard to detect.

This paper presents two contributions. First, we present an approach that automatically infers likely changes at or above the level of method headers, and uses this information to determine method-level matches. Second, our approach

represents the inferred changes concisely as first-order relational logic rules, each of which combines a set of similar low-level transformations and describes exceptions that capture anomalies to a general change pattern. Explicitly representing exceptions to a general change pattern makes our algorithm more robust because a few exceptions do not invalidate a high-level change, and it can signal incomplete or inconsistent changes as likely design errors.

For the preceding change scenario, our tool infers a rule—details appear in Section 3—for each of the high-level changes made by the programmer.

```
for all x in chart.*Axis*.*(*)
  packageReplace(x, chart, chart.axis)
for all x in chart.Factory.create*Chart(*Data)
  except {createGanttChart, createXYChart}
  argAppend(x, [boolean])
```

We applied our tool to several open source projects. In terms of matching, our evaluation shows that our tool finds matches that are hard to find using other tools. Our rule representation makes results smaller and more readable. We also believe that by capturing changes in a concise and comprehensible form, our technique may enable software engineering applications that can benefit from high level change patterns; for example, bug detectors, documentation assistance tools, and API update engines. Moreover, the mining of software repositories can be enhanced by the accuracy of our algorithm and the information captured by our rules.

Section 2 discusses related work. Section 3 describes the representation of the structural changes our tool infers. Section 4 describes our inference algorithm. Section 5 presents our results and compares our approach to other approaches. Section 6 discusses potential applications of our change rules based on motivating examples from our study. Section 7 discusses future work.

2. Background

Program Element Matching Techniques. To match code elements, the differences between two programs must be identified. Computing semantic differences is undecidable, so tools typically approximate matching via syntactic similarity. Tools differ in their underlying program representation, matching granularity, matching multiplicity, and matching heuristics. In prior work [23], we compared existing matching techniques [3, 5, 17, 18, 19, 20, 24, 26, 28, 32, 35, 37] along these dimensions. Our survey found that fine-grained matching techniques often depend on effective mappings at a higher level. For example, *Jdiff* [3] cannot match control flow graph nodes if function names are very different. Therefore, when package level or class level refactorings occur, these techniques will miss many matches. Our survey also found that most techniques work at a fixed granularity and that most techniques report only one-to-one mappings between code elements in the version pair. These properties limit the techniques in the case of merging or splitting, which are commonly performed by programmers. Most existing tools report their results as an unstructured list of matches, further limiting the potential of these approaches.

Origin analysis tools [24, 39] find where a particular code element came from, tackling the matching problem directly. Zou and Godfrey's analysis [39] matches procedures using multiple criteria (names, signatures, metric values, and callers/callees). It is semi-automatic; a programmer must manually tune the matching criteria and select a match among candidate matches. S. Kim et al. [24] automated Zou and Godfrey's analysis; matches are automatically selected if an overall similarity, computed as a weighted sum of the underlying similarity metrics, exceeds a given threshold.

Refactoring Reconstruction. Existing refactoring reconstruction tools [2, 8, 9, 11, 21, 25, 31, 33, 34] compare code between two program versions and look for code changes that match a predefined set of refactoring patterns: move a method, rename a class, etc. For example, UMLDiff [34] matches code elements in a top-down order, e.g., packages to classes to methods, based on name and structural similarity metrics. Then it infers refactorings from the matched code elements. As another example, Fluri et al. [11] compute tree edit operations between two abstract syntax trees and identify changes inside the same class.

Many of these tools either find too many refactoring candidates and cannot disambiguate which ones are more likely than others, or they do not find some refactorings when some code elements undergo multiple refactorings during a single check-in. Like matching tools, many of these tools report simply a list of refactorings, making it difficult to find emerging change patterns or to discover insights about why a particular set of refactorings may have occurred.

3. Change Rules

Determining the changes between two versions enables matching of their code elements. The research question is: *"Given two versions of a program, what changes occurred with respect to a particular vocabulary of changes?"* A change vocabulary characterizes the applications for which the matching results can be used. For example, *diff* defines a change vocabulary in terms of delete, add, and move line. Though this change vocabulary is satisfactory for applications such as conventional version control, it may not be ideal for studying program evolution.

Consider Table 1. One change moves a group of classes that match the `*Axis` pattern from package `chart` to package `chart.axis`, and the other change adds a `boolean` parameter to a group of methods that match the `create*Chart`

Table 1. Comparison between Programmer's Intent and Existing Tools' Results

Programmer's Intent	Matching Tool Results [24]	Refactoring Reconstruction Results [33]

Move classes that draw axes from `chart` package to `chart.axis` package	`[chart.DateAxis..., chart.axis.DateAxis...]`
	`[chart.NumberAxis..., chart.axis.NumberAxis...]`
	`[chart.ValueAxis..., chart.axis.ValueAxis...]`
		Move class `DateAxis` from `chart` to `chart.axis`
		Move class `NumberAxis` from `chart` to `chart.axis`
		Move class `ValueAxis` from `chart` to `chart.axis`

Widen the APIs of chart factory methods by adding a `boolean` type argument	`[createAreaChart(Data), createAreaChart(Data, boolean)]`
	`[createLChart(IntData), createLChart(IntData, boolean)]`
	`[createPieChart(PieData), createPieChart(PieData, boolean)]`
		Add `boolean` parameter to `createAreaChart`
		Add `boolean` parameter to `createLChart`
		Add `boolean` parameter to `createPieChart`

pattern. Both changes involve applying the same atomic change to a set of related code elements.

We have developed a change vocabulary with the goal of representing groups of related homogeneous changes precisely. The core of our change vocabulary is a change rule consisting of a quantifier and a scope to which a low-level transformation applies: "for all x in (scope $-$ exceptions), $t(x)$," where t is a transformation, *scope* is a set of code elements, and *exceptions* is a subset of *scope*.

Currently, our change vocabulary describes structural changes at or above the level of a method header. Given two versions of a program, (P_1, P_2), our goal is to find a set of *change rules*, i.e., change functions that transform the method headers in P_1 to generate the method headers in P_2, in turn generating a set of method-level matches from these change functions. In this paper, a Java method header is represented as *`return_type package.class.procedure(input_argument_list)`*.[1]

Transformation. Currently we support the following types of transformations:

- *packageReplace(x:Method, f:Text, t:Text)*:
  change x's package name from f to t
- *classReplace(x:Method, f:Text, t:Text)*:
  change x's class name from f to t
- *procedureReplace(x:Method, f:Text, t:Text)*:
  change x's procedure name from f to t
- *returnReplace(x:Method, f:Text, t:Text)*:
  change x's return type from f to t
- *inputSignatureReplace(x:Method,f:List[Text], t:List[Text])*:
  change x's input argument list from f to t
- *argReplace(x:Method, f:Text, t:Text)*:
  change argument type f to t in x's input argument list
- *argAppend(x:Method, t:List[Text])*:
  append the arg type list t to x's input argument list
- *argDelete(x:Method, t:Text)*:
  delete every occurrence of type t in the x's input argument list
- *typeReplace(x:Method, f:Text, t:Text)*:
  change every occurrence of type f to t in x

[1]The return_type is sometimes omitted for presentation purposes.

Rule. A change rule consists of a *scope*, *exceptions* and a *transformation*. The only methods transformed are those in the scope but not in the exceptions. When a group of methods have similar names, we summarize these methods as a scope expression using a wild-card pattern matching operator. For example, *`*.*Plot.get*Range()`* describes methods with any package name, any class name that ends with `Plot`, any procedure name that starts with `get` and ends with `Range`, and an empty argument list.

To discover emerging transformation patterns, a scope can have disjunctive scope expressions. The following rule means that all methods whose class name either includes `Plot` or `JThermometer` changed their package name from `chart` to `chart.plot`.

```
for all x in chart.*Plot*.*(*)
  or chart.*JThermometer*.*(*)
  packageReplace(x, chart, chart.plot)
```

As another example, the following rule means that all methods that match the *`chart.*Plot.get*Range()`* pattern take an additional `ValueAxis` argument, except the `getVerticalRange` method in the `MarkerPlot` class.

```
for all x in chart.*Plot.get*Range()
  except {chart.MarkerPlot.getVerticalRange}
  argAppend(x, [ValueAxis])
```

Rule-based Matching. We define a matching between two versions of a program by a set of change rules. The scope of one rule may overlap with the scope of another rule as some methods undergo more than one transformation. Our algorithm ensures that we infer a set of rules such that the application order of rules does not matter. The methods that are not matched by any rules are deleted or added methods. For example, the five rules in Table 2 explain seven matches. The unmatched method O2 is considered deleted.

4. Inference Algorithm

Our algorithm accepts two versions of a program and infers a set of change rules. Our algorithm has four parts: (1) generating seed matches, (2) generating candidate rules based on the seeds, (3) iteratively selecting the best rule among the candidate rules, and (4) post-processing the selected candidate rules to output a set of change rules. We first describe a naïve version of our algorithm, followed by a description of essential performance improvements for the second and third parts of the algorithm. Then we summarize

Table 2. Rule-based Matching Example

A set of method headers in P_1	A set of method headers in P_2
O1. `chart.VerticalPlot.draw(Grph,Shp)`	N1. `chart.plot.VerticalPlot.draw(Grph)`
O2. `chart.VerticalRenderer.draw(Grph,Shp)`	N2. `chart.plot.HorizontalPlot.range(Grph)`
O3. `chart.HorizontalPlot.range(Grph,Shp)`	N3. `chart.axis.HorizontalAxis.getHeight()`
O4. `chart.HorizontalAxis.height()`	N4. `chart.axis.VerticalAxis.getHeight()`
O5. `chart.VerticalAxis.height()`	N5. `chart.ChartFactory.createAreaChart(Data, boolean)`
O6. `chart.ChartFactory.createAreaChart(Data)`	N6. `chart.ChartFactory.createGanttChart(Interval, boolean)`
O7. `chart.ChartFactory.createGanttChart(Interval)`	N7. `chart.ChartFactory.createPieChart(PieData, boolean)`
O8. `chart.ChartFactory.createPieChart(PieData)`	

Rule			Matches Explained
scope	*exceptions*	*transformation*	
chart.*Plot.*(*)		***packageReplace(x, chart, chart.plot)***	[O1, N1], [O3, N2]
chart.*Axis.*(*)		***packageReplace(x, chart, chart.axis)***	[O4, N3], [O5, N4]
chart.ChartFactory.create*Chart(*)		***argAppend(x, [boolean])***	[O6, N5], [O7, N6], [O8, N7]
chart.*.*(Grph, Shp)	{O2}	***argDelete(x, Shp)***	[O1, N1], [O3, N2]
chart.*Axis.height()		***procedureReplace(x, height, getHeight)***	[O4, N3], [O5, N4]

key characteristics of our algorithm.

Part 1. Generating Seed Matches. We start by searching for method headers that are similar on a textual level, which we call *seed matches*. Seed matches provide initial hypotheses about the kind of changes that occurred. Given the two program versions (P_1, P_2), we extract a set of method headers O and N from P_1 and P_2 respectively. Then, for each method header x in $O - N$, we find the closest method header y in $N - O$ in terms of the token-level name similarity, which is calculated by breaking x and y into a list of tokens starting with capital letters and then computing the longest common subsequence of tokens [18]. If the name similarity is over a threshold γ, we add the pair into the initial set of seed matches. In our study, we found that thresholds in the range of 0.65-0.70 (meaning 65% to 70% of tokens are the same) gave good empirical results. The seeds need not all be correct matches, as our rule selection algorithm (Part 3) rejects bad seeds and leverages good seeds. Seeds can instead come from other sources such as CVS comments, other matching tools, or recorded or inferred refactorings.

Part 2. Generating Candidate Rules. For each seed match $[x, y]$, we build a set of *candidate rules* in three steps. Unlike a change rule, where for every match $[x, y]$, y is the result of applying a single transformation to x, a candidate rule may include one or more transformations $t_1, \ldots t_i$ such that $y = t_1(...t_i(x))$. We write candidate rules as "for all x in scope, $t_1(x) \wedge ... \wedge t_i(x)$." This representation allows our algorithm to find a match $[x, y]$ where x undergoes multiple transformations to become y.

Step 1. We compare x and y to find a set of transformations T= $\{t_1, t_2, \ldots, t_i\}$ such that $t_1(t_2(\ldots t_i(x)))$= y. We then create T's power set 2^T. For example, a seed [`chart.VerticalAxis.height()`, `chart.plot.VerticalAxis.getHeight()`] produces the power set of *`packageReplace(x, chart, chart.plot)`* and *`procedureReplace(x, height, getHeight)`*.

Step 2. We guess scope expressions from a seed match $[x, y]$. We divide x's full name to a list of tokens starting with capital letters. For each subset, we replace every token in the subset with a wild-card operator to create a candidate scope expression. As a result, when x consists of n tokens, we generate a set of 2^n scope expressions based on x. For the preceding example seed, our algorithm finds S =$\{$*`.*.*(*)`, *`chart.*.*(*)`*, *`chart.Vertical*.*(*)`*, ..., *`*.*Axis.height()`*, ..., *`chart.VerticalAxis.height()`*$\}$.

Step 3. We generate a candidate rule with scope expression s and compound transformation t for each (s, t) in $S \times 2^T$. We refer to the resulting set of candidate rules as CR. Each element of CR is a generalization of a seed match and some are more general than others.

Part 3. Evaluating and Selecting Rules. Our goal is to select a small subset of candidate rules in CR that explain a large number of matches. While selecting a set of candidate rules, we allow candidate rules to have a limited number of exceptions to the general rule they represent.

The inputs are a set of candidate rules (CR), a domain ($D = O - N$), a codomain ($C = N$), and an exception threshold ($0 \leq \epsilon < 1$). The outputs are a set of selected candidate rules (R), and a set of found matches (M). For a candidate rule r, "for all x in scope, $t_1(x) \wedge ... \wedge t_i(x)$":

1. r has a **match** $[a, b]$ if $a \in$ scope, $t_1, ..., t_i$ are applicable to a, and $t_1(...t_i(a)) = b$.
2. a match $[a, b]$ **conflicts** with a match $[a', b']$ if $a = a'$ and $b \neq b'$
3. r has a **positive** match $[a, b]$ given D, C, and M, if $[a, b]$ is a match for r, $[a, b] \in \{D \times C\}$, and none of the matches in M conflict with $[a, b]$
4. r has a **negative** match $[a, b]$, if it is a match for r but not a positive match for r.
5. r is a **valid** rule if the number of its positive matches is at least $(1 - \epsilon)$ times its matches. For example, when ϵ is 0.34 (our default), r's negative matches (exceptions) must be fewer than roughly one third of its matches.

Our algorithm greedily selects one candidate rule at each iteration such that the selected rule maximally increases the

total number of matches. Initially we set both R and M to the empty set. In each iteration, for every candidate rule $r \in CR$, we compute r's matches and check whether r is valid. Then, we select a valid candidate rule s that maximizes $|M \cup P|$ where P is s's positive matches. After selecting s, we update $CR := CR - \{s\}$, $M := M \cup P$, and $R := R \cup \{(s, P, E)\}$ where P and E are s's positive and negative matches (exceptions) respectively, and we continue to the next iteration. The iteration terminates when no remaining candidate rules can explain any additional matches. The naïve version of this greedy algorithm has $O(|CR|^2 \times |D|)$ time complexity.

Part 4. Post Processing. To convert a set of candidate rules to a set of rules, for each transformation t, we find all candidate rules that contain t and then create a new scope expression by combining these rules' scope expressions. Then we find exceptions to this new rule by enumerating negative matches of the candidate rules and checking if the transformation t does not hold for each match.

Optimized Algorithm. Two observations allow us to improve the naïve algorithm's performance. First, if a candidate rule r can add n additional matches to M at the i^{th} iteration, r cannot add more than n matches on any later iteration. By storing n, we can skip evaluating r on any iteration where we have already found a better rule s that can add more matches than r. Second, candidate rules have a subsumption structure because the scopes can be subsets of other scopes (e.g., `*.*.*(*Axis)` $\subset$ `*.*.*(*)`).

Our optimized algorithm behaves as follows. Suppose that the algorithm is at the i^{th} iteration, and after examining $k-1$ candidate rules in this iteration, it has found the best valid candidate rule s that can add N additional matches. For the k^{th} candidate rule r_k,
(1) If r_k could add fewer than N additional matches up to i-1^{st} iteration, skip evaluating r_k as well as candidate rules with the same set of transformations but a smaller scope, as our algorithm does not prefer r_k over s.
(2) Otherwise, reevaluate r_k.

(2.1) If r_k cannot add any additional matches to M, remove r_k from CR.

(2.2) If r_k can add fewer than N additional matches regardless of its validity, skip evaluating candidate rules with the same set of transformations but a smaller scope.

(2.3) If r_k is not valid but can add more than N additional matches to M, evaluate candidate rules with smaller scope and the same set of transformations.
(3) Update s and N as needed and go to step (1) to consider the next candidate rule in CR.

By starting with the most general candidate rule for each set of transformations and generating more candidate rules on demand only in step (2.3) above, the optimized algorithm is much more efficient. Running our tool currently takes only a few seconds for the usual check-ins and about seven minutes in average for a program release pair.

Key Characteristics of Our Algorithm. First, our algorithm builds insight from seed matches, generalizes the scope that a transformation applies to, and validates this insight. Second, it prefers a small number of general rules to a large number of specific rules. Third, when there are a small number of exceptions that violate a general rule, our algorithm allows these exceptions but remembers them.

5. Evaluation

In this section, we quantitatively evaluate our tool in terms of its matching power. Then in the following section, we qualitatively assess additional benefits of inferred rules based on examples found in our study. To evaluate our matches (M), we created a set of correctly labeled matches (E). We did this in two steps. First, we used our own inference algorithm on each version pair in both directions (which can find additional matches) and we computed the union of those matches with the matches found by other approaches. Second, we labeled correct matches through a manual inspection. Our quantitative evaluation is based on the three following criteria.

Precision: the percentage of our matches that are correct, $\frac{|E \cap M|}{|M|}$.

Recall: the percentage of correct matches that our tool finds, $\frac{|M \cap E|}{|E|}$.

Conciseness: the measure of how concisely a set of rules explains matches, represented as a M/R ratio = $\frac{|M|}{|Rules|}$. A high M/R ratio means that using rules instead of plain matches significantly reduces the size of results.

Our evaluations are based on both releases (i.e., released versions) as well as check-in snapshots (i.e., internal, intermediate versions). The primary difference is that there tends to be a much larger delta between successive program releases than between successive check-in snapshots.

Section 5.1 presents rule-based matching results for three open source release archives. Sections 5.2 presents comparison with two refactoring reconstruction tools [33, 34] and a method-level origin analysis tool [24]. Section 5.3 discusses the impact of the seed generation threshold (γ) and the exception threshold (ϵ). Section 5.4 discusses threats to the validity of our evaluation.

5.1. Rule-Based Matching Results

Subject Programs. We chose three open source Java programs that have release archives on *sourceforge.net* and contain one to seven thousand methods. The moderate size lets us manually inspect matches when necessary. *JFreeChart* is a library for drawing different types of charts,

Table 3. Rule-based Matching Results

	O	N	O ∩ N	Rule	Match	Prec.	Recall	M/R	Time
JFreeChart (www.jfree.org/jfreechart) The actual release numbers are prefixed with 0.9.									
4→5	2925	3549	1486	178	1198	0.92	0.92	6.73	21.01
5→6	3549	3580	3540	5	6	1.00	1.00	1.20	<0.01
6→7	3580	4078	3058	23	465	1.00	0.99	20.22	1.04
7→8	4078	4141	0	30	4057	1.00	0.99	135.23	43.06
8→9	4141	4478	3347	187	659	0.91	0.90	3.52	22.84
9→10	4478	4495	4133	88	207	0.99	0.93	2.35	0.96
10→11	4495	4744	4481	5	14	0.79	0.79	2.80	<0.01
11→12	4744	5191	4559	61	113	0.78	0.79	1.85	0.40
12→13	5191	5355	5044	10	145	1.00	0.99	14.50	0.11
13→14	5355	5688	5164	41	134	0.94	0.86	3.27	0.43
14→15	5688	5828	5662	9	21	0.90	0.70	2.33	0.01
15→16	5828	5890	5667	17	77	0.97	0.86	4.53	0.32
16→17	5890	6675	5503	102	285	0.91	0.86	2.79	1.30
17→18	6675	6878	6590	10	61	0.90	1.00	6.10	0.08
18→19	6878	7140	6530	98	324	0.93	0.95	3.31	1.67
19→20	7140	7222	7124	4	14	1.00	1.00	3.50	<0.01
20→21	7222	6596	4454	71	1853	0.99	0.98	26.10	62.99
MED						**0.94**	**0.93**	**3.50**	**0.43**
MIN						**0.78**	**0.70**	**1.20**	**0.00**
MAX						**1.00**	**1.00**	**135.23**	**62.99**
JHotDraw (www.jhotdraw.org)									
5.2→5.3	1478	2241	1374	34	82	0.99	0.92	2.41	0.11
5.3→5.41	2241	5250	2063	39	104	0.99	0.98	2.67	0.71
5.41→5.42	5250	5205	5040	17	17	0.82	1.00	1.00	0.07
5.42→6.01	5205	5205	0	19	4641	1.00	1.00	244.26	27.07
MED						**0.99**	**0.99**	**2.54**	**0.41**
MIN						**0.82**	**0.92**	**1.00**	**0.07**
MAX						**1.00**	**1.00**	**244.26**	**27.07**
jEdit (www.jedit.org)									
3.0→3.1	3033	3134	2873	41	63	0.87	1.00	1.54	0.13
3.1→3.2	3134	3523	2398	97	232	0.93	0.98	2.39	1.51
3.2→4.0	3523	4064	3214	102	125	0.95	1.00	1.23	0.61
4.0→4.1	4064	4533	3798	89	154	0.88	0.95	1.73	0.90
4.1→4.2	4533	5418	3799	188	334	0.93	0.97	1.78	4.46
MED						**0.93**	**0.98**	**1.73**	**1.21**
MIN						**0.87**	**0.95**	**1.23**	**0.61**
MAX						**0.95**	**1.00**	**2.39**	**4.46**

JHotDraw is a GUI framework for technical and structured graphics, and *jEdit* is a cross platform text editor. On average, release versions were separated by a two-month gap in *JFreeChart* and a nine-month gap in *JHotDraw* and *jEdit*.

Results. Table 3 summarizes results for the projects (γ=0.7 and ϵ=0.34). O and N are the number of methods in an old version and a new version respectively, and $O \cap N$ is the number of methods whose name and signature did not change. Running time is described in minutes.

The precision of our tool is generally high in the range of 0.78 to 1.00, and recall is in the range 0.70 to 1.00. The median precision and the median recall for each set of subjects is above, often well above, 0.90.

The M/R ratio shows significant variance not only across the three subjects but also for different release pairs in the same subject. The low end of the range is at or just over 1 for each subject, representing cases where each rule represents roughly a single match. The high end of the range varies from 2.39 (for *JEdit*) to nearly 244.26 (for *JHotDraw*). We observed, however, that most matches are actually found by a small portion of rules (recall our algorithm finds rules in descending order of the number of matches).

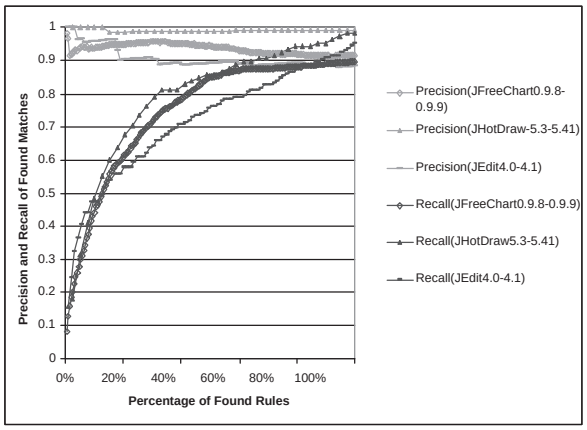

Figure 1. Recall and Precision vs. Percentage of Found Matches

Figure 1 plots the cumulative distribution of matches for the version pairs with the median M/R ratio from each of the three projects. The x axis represents the percentage of rules found after each iteration, and the y axis represents the recall and precision of matches found up to each iteration.

In all three cases, the top 20% of the rules find over 55% of the matches, and the top 40% of the rules find over 70% of the matches. In addition, as the precision plots show, the matches found in early iterations tend to be correct matches evidenced by a general change pattern. The fact that many matches are explained by a few rules is consistent with the view that a single conceptual change often involves multiple low level transformations, and it confirms that leveraging an emergent change structure is a good matching approach.

Our tool handled the major refactorings in the subject programs quite well. For example, consider the change from release 4 to 5 of *JFreeChart*. Although nearly half of the methods cannot be matched by name, our tool finds 178 rules and 1198 matches. The inferred rules indicate that there were many package-level splits as well as low-level API changes. As presented below, these kind of changes are not detected by other tools we analyzed. Examples of the inferred rules in *JFreeChart* include:

```
for all x in chart.*Plot.*(CategoryDataSet)
   or chart.*.*(Graph, Rect, Rect2D)
   or chart.*.*(Graph, Plot, Rect2D)
   argAppend(x, [int])
for all x in int renderer.*.draw*(*, Graph, Rect)
   returnReplace(int, AxisState)
```

5.2. Comparison with Other Approaches

Refactoring reconstruction tools [2, 8, 9, 33, 34] compare two versions of a program and look for code changes that match a predefined set of refactoring patterns [12].

Table 4. Comparison: Number of Matches and Size of Result

Other Approach				Our Approach		Improvement			
Xing and Stroulia (XS)		**Match**	**Refactoring**	**Match**	**Rules**				
jfreechart	(17 release pairs)	8883	4004	9633	939	8%	more matches	77%	decrease in size
Weigerber and Diehl (WD)		**Match**	**Refactoring**	**Match**	**Rules**				
jEdit	*RCAll*	1333	2133	1488	906	12%	more matches	58%	decrease in size
(2715 check-ins)	*RCBest*	1172	1218	1488	906	27%	more matches	26%	decrease in size
Tomcat	*RCAll*	3608	3722	2984	1033	17%	fewer matches	72%	decrease in size
(5096 check-ins)	*RCBest*	2907	2700	2984	1033	3%	more matches	62%	decrease in size
S. Kim et al (KPW)		**Match**		**Match**	**Rules**				
jEdit	(1189 check-ins)	1430		2009	1119	40%	more matches	22%	decrease in size
ArgoUML	(4683 check-ins)	3819		4612	2127	21%	more matches	44%	decrease in size

Table 5. Comparison: Precision

Comparison of Matches			Match	Precision
Xing and Stroulia (XS)		XS ∩ Ours	8619	1.00
jfreechart		Ours − XS	1014	0.75
(17 release pairs)		XS − Ours	264	0.75
Weißgerber and Diehl (WD)		WD ∩ Ours	1045	1.00
jEdit (2715 check-ins)	*RCAll*	Ours − WD	443	0.94
		WD − Ours	288	0.36
	RCBest	WD ∩ Ours	1026	1.00
		Ours − WD	462	0.93
		WD − Ours	146	0.42
Tomcat (5096 check-ins)	*RCAll*	WD ∩ Ours	2330	0.99
		Ours − WD	654	0.66
		WD − Ours	1278	0.32
	RCBest	WD ∩ Ours	2251	0.99
		Ours − WD	733	0.75
		WD − Ours	656	0.54
S. Kim et al. (KPW)		KPW ∩ Ours	1331	1.00
jEdit (1189 check-ins)		Ours − KPW	678	0.89
		KPW − Ours	99	0.75
ArgoUML (4683 check-ins)		KPW ∩ Ours	3539	1.00
		Ours − KPW	1073	0.78
		KPW − Ours	280	0.76

Among these tools, we compared our matching results with Xing and Stroulia's approach (XS) [34] and Weißgerber and Diehl's approach (WD) [33]. To uniformly compare our rules with the results of XS and WD's approaches, we built a tool that deduces method-level matches from their inferred refactorings. Then we compared both approaches in terms of the number of matches as well as the size of the results (the number of rules in our approach and the number of relevant refactorings in XS and WD's approach).

Among origin analysis tools, we chose S. Kim et al.'s approach (KPW) [24] for comparison because it is the most recent work that we know of and it reported 87.8% to 91.1% accuracy on their evaluation data set.[2]

For comparison, XS provided their results on *JFreeChart* release archives, WD provided their results on *jEdit* and *Tomcat* check-in snapshots, and KPW provided their results on *jEdit* and *ArgoUML* check-in snapshots.

Comparison with Xing and Stroulia's UMLDiff. XS's tool UMLDiff extracts class models from two versions of a program, traverses the two models, and identifies corresponding entities based on their name and structure similarity. Then it reports additions and removals of these entities and inferred refactorings. XS can find most matches that involve more than one refactoring; however, to reduce its computational cost, it does not handle combinations of move and rename refactorings such as 'move `CrosshairInfo` class from `chart` to `chart.plot` package' and 'rename it to `CrosshairState`.'

The comparison results are summarized in Tables 4 and 5. Overall, XS's precision is about 2% (=8807/8883-9369/9633) higher. However, our tool finds 761 (=1014×0.75) correct matches not found by XS while there are only 199 (=264×0.75) correct matches that our tool failed to report. More importantly, our tool significantly reduces the result size by 77% by describing results as rules. Many matches that XS missed were matches that involve both rename and move refactorings. Many matches that our tool missed had a very low name similarity, indicating a need to improve our current seed generation algorithm.

Comparison with Weißgerber and Diehl's Work. WD's tool extracts added and deleted entities (fields, methods and classes) by parsing deltas from a version control system (CVS) and then compares these entities to infer various kinds of structural and local refactorings: move class, rename method, remove parameter, etc. The tool finds redundant refactoring events for a single match. For example, if the `Plot` class were renamed to `DatePlot`, it would infer 'rename class `Plot` to `DatePlot` as well as 'move method' refactorings for all methods in the `Plot` class. When it cannot disambiguate all refactoring candidates, it uses the clone detection tool CCFinder [20] to rank these refactorings based on code similarity. For example, if `VerticalPlot.draw(Graph)` is deleted and `VerticalPlot.drawItem(Graph)` and `VerticalPlot.render(Graph)` are added, it finds both 'rename method `draw` to `drawItem`' and 'rename method `draw` to `render`,' which are then ordered.

We compared our results both with (1) all refactoring candidates RC_{all} and (2) only the top-ranked refactoring candidates RC_{best}. The comparison results with RC_{best} and RC_{all} (γ=0.65 and ϵ=0.34) are shown in Table 4 and 5. When comparing with RC_{best}, our approach finds 27% more matches yet decreases the result size by 26% in *jEdit*, and it finds 3% more matches yet decreases the result size

[2]KPW created an evaluation data by having human judges identify renaming events in *Subversion* and *Apache* projects. The accuracy is defined as the percentage of agreement between two sets of origin relations.

by 62% in *Tomcat*. This result shows our approach achieves better matching coverage while retaining concise results. We also compared our matches and the matches generated by WD's tool. We manually inspected 50 sample check-ins to estimate precision for the matches missed by one tool but not the other as well as the matches found by both tools. For *jEdit*, our approach found 462 matches not identified by WD's RC_{best}, and RC_{best} found just over 146 matches that we failed to report. When combined with the precision, this means our approach found about 430 (=462×0.93) additional useful matches, and their approach found about 61 (=146×0.42) additional useful matches. *Tomcat* shows roughly similar results. WD's tool missed many matches when compound transformations were applied. Our tool missed some matches because γ=0.65 did not generate enough seeds to find them.

Comparison with S. Kim et al.'s Origin Analysis. For comparison, both our tool and KPW's tool were applied to *jEdit* and *ArgoUML*'s check-in snapshots. Table 4 and 5 shows the comparison result (γ=0.65 and ϵ=0.34). For *jEdit*, our approach finds 40% more matches yet reduces the result size by 22%, and for *ArgoUML*, it finds 21% more matches yet reduces the result size by 44%.

We also compared our matches to KPW's matches and inspected the matches from 50 sample check-ins to measure precision. For *jEdit*, we found over 678 matches not identified by KPW's approach, and KPW's approach found about 99 matches that we did not. When combined with the precision of sampled matches, this means our approach found over 600 (=678×0.89) useful matches and that KPW's approach found about 75 (=99×0.75) useful matches. *ArgoUML* shows roughly similar results. This result is noteworthy because KPW's approach considers more information such as calling relationships as well as clone detection results in addition to name similarity. We suspect that it is because KPW's approach cannot accept correct matches when their overall similarity score is lower than a certain threshold and cannot easily prune incorrect matches once their overall similarity score is over a certain threshold and is higher than other matches. On the other hand, our algorithm tends to reject matches whose transformation is an isolated incident even if the similarity score is high. Our tool's incorrect matches usually come from bad seeds that coincidentally have similar names. Overall, our approach finds more matches without sacrificing its precision and represents results more concisely than KPW's approach.

5.3. Impact of Threshold

Seed Threshold. Our results in part depend on the quantity and quality of seeds. Figure 2 shows how our algorithm behaves when we change the seed generation threshold γ for *JFreechart* (0.9.4→0.9.5). We varied γ from 0.9 to 0.5

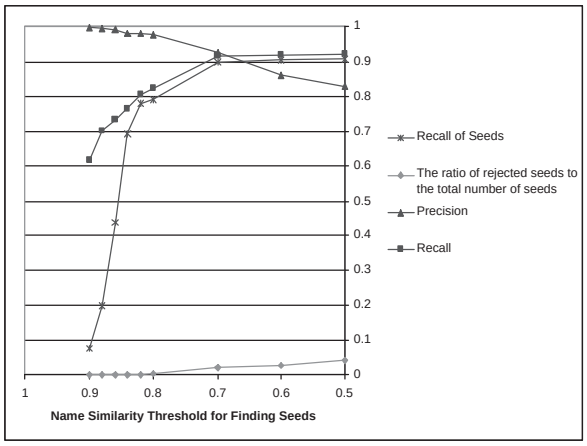

Figure 2. Impact of Seed Threshold γ

and measured recall of seeds, precision, recall, and the ratio of rejected seeds to the total number of seeds. When γ is set high in the range of 0.9 to 0.8, the name matching technique finds a relatively small number of seeds, but the seeds tend to be all good seeds. So our algorithm rejects very few seeds and leverages the good seeds to quickly reach the recall of 0.65 to 0.85. However, the recall is still below 0.85 as the seeds do not contain enough transformations. As γ decreases, more seeds are produced and a higher percentage of them are bad seeds that our algorithm later rejects. Using a low threshold (< 0.6) generally leads to higher recall (above 0.9) but lowers precision and increases the running time since there are more candidate rules based on bad seeds. For the results in Figure 2, we observed a roughly linear increase from 6 minutes (γ=0.9) to 26 minutes (γ=0.5).

In general, when the precision and recall of seed matches are low, our algorithm improves both measures significantly. When the seed matches already have precision and recall over 0.9, the algorithm still improves both measures, although less so because the seeds are already very good. However, even in this case, our algorithm significantly improves the conciseness measure. Effective seed generation and its interaction with our candidate rule selection algorithm needs additional research.

Exception Threshold. We experimented with different exception thresholds: 0.25, 0.34, 0.5. Using a low threshold increases running time and slightly decreases the M/R ratio. Surprisingly we found that changing exception thresholds does not affect precision and recall much. We suspect that it is because most exceptions come from deleted entities.

5.4. Threats to Validity

To measure precision, the first author manually inspected the matches generated by our tool and by other tools. Man-

ual labeling is subject to evaluator bias. All data are publicly available,[3] so other researchers can independently assess our results (and use our data).

Our effort to date is limited in a number of ways. First, we have not explored other (non-Java) programming languages, other (non-object-oriented) programming paradigms, or even different naming conventions, all of which could have consequences. Second, we have not explored possible ways to exploit information from programming environments (such as Eclipse) that support higher-level operations, such as some common refactorings.

6. Applications of Change Rules

Our rules represent inferred changes in a concise and comprehensible form. This allows them to serve as a basis for many software engineering applications that can benefit from additional knowledge about change. We sketch several such applications and include motivating examples from our study. (Some of the example rules below are slightly modified for presentation purposes.)

Bug Finding. While examining the inferred rules, we found that exceptions often signal a bug arising from incomplete or inconsistent changes. For example, the rule

```
for all x in J*.addTitle(*)
  except { JThermometer.addTitle(Title) }
  procedureReplace(x, addTitle, addSubtitle)
```

has one exception, which indicates that a programmer misspelled `addSubtitle` to `addSubitle` when modifying the `addTitle` method of `JThermometer`, which is a subclass of `JFreeChart`. This misspelling causes dynamic dispatching to `JThermometer` not to function properly because `addSubtitle` is no longer overridden.

As another example, consider the following two rules, in which the second one is found one release after the first one. We suspect that a programmer fixed only two out of the three problems, leaving one bug.

```
for all x in *.draw(*, Key, *)
  except { HorizontalBar, VerticalBar, StatBar }
  argReplace(x, Key, Category)
for all x in *Bar.draw(*, Key, *)
  except { VerticalBar }
  argReplace(x, Key, Category)
```

A similar idea that detects potential errors from inferred refactorings has been explored by Görg and Weißgerber [14]. However, they check only a predefined set of refactoring consistency patterns.

Assisted Documentation. By inferring change rules at each check-in, our approach could be integrated into a version control system to assist programmers in documenting their changes. Suppose that a programmer modified 14 methods to modify plot drawing APIs. Our tool can infer the following rule, summarizing the changes concisely.

```
for all x in chart.plot.*Plot.draw(ChartInfo)
  inputSignatureReplace([ChartInfo],
  [PlotState,PlotInfo])
```

If a programmer wants to examine instances of this change, we can simply display the matches found by the rule.

If inferring and documenting rules becomes a part of standard development practice, this practice opens doors for richer software evolution research. In classic software evolution studies [6], changes are often measured in terms of LOC or the number of components. These quantitative metrics do not necessarily depict an accurate picture of evolution in general. For the preceding example, while measuring LOC changes may show that there were hundreds of lines of dispersed changes, our rule can convey qualitative information about evolution that conventional metrics cannot.

API Evolution Analysis. Our inferred rules may be able to shed light on understanding the evolution of APIs [10]. Suppose that a programmer removed the `Shape` type in some APIs to hide unnecessary details from clients. Before checking in this change, our tool can automatically infer the following rule and assist in describing what kinds of details are hidden:

```
for all x in chart.*.*(Graphic, *, Shape)
  argDelete(x, Shape)
```

Someone who sees this comment later will better understand the intention of the change. More importantly, we found our inferred rules often reveal volatility of some API changes. In the following example, the first rule shows that the use of `Category` type was replaced by `[Key, int]` type.

```
for all x in *.*.*(Category)
  inputSignatureReplace(x,[Category],[Key, int])
for all x in *.*.*(Key, int)
  inputSignatureReplace(x,[Key, int],[Category])
```

In the next release, we found that that the same change was quickly reversed based on the second rule.

API Update. When an imported API is modified, programmers often have to update their code manually to adapt to the modified interface. One approach, embodied by CatchUp! captures refactorings performed within Eclipse and uses this information to help API users to update their code accordingly [7, 16]. In the absence of recorded refactorings, our tool can act as a refactoring reconstruction tool [9, 33], feeding input for an API update tool.

Identifying Related Changes. Recent work has proposed ways to use historical information to identify parts of a software system that are likely to change together [13, 36, 38]. By grouping related changes and representing them as a rule, our tool can also identify what parts of

[3] www.cs.washington.edu/homes/miryung/matching

a system change together. For example, consider this rule with 13 matches.

```
for all x in int *.*.*(CharIndex, REMatch)
   returnReplace(x, int, boolean)
```

This rule shows a programmer modified a group of methods similarly although they are not in the same class or package. If one wants to factor out a set of related but scattered methods using the AspectJ programming language [22], we believe that the scope expressions in our rules can be excellent candidates for the point-cut descriptor.

7. Conclusions

Our approach is the first to automatically infer structural changes, represent them concisely as a set of rules, and use the rules to determine matches between two program versions. Our tool finds more and better matches than the other tools we surveyed and evaluated. Furthermore, the inferred change rules show promise in enabling new approaches to a wide variety of software engineering applications.

Acknowledgments. We especially thank Sunghun Kim, Peter Weißgerber and Zhenchang Xing for providing their data and thank Görel Hedin, Martin Robillard, Vibha Sazawal, Annie Ying, Laura Effinger-Dean, Ben Lerner, and the anonymous reviewers for comments on our draft.

References

[1] MSR'06: Proceedings of the 2006 int'l workshop on mining software repositories. General Chair-Stephan Diehl and Harald Gall and Ahmed E. Hassan.

[2] G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *IWPSE*, pages 31–40, 2004.

[3] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *ASE*, pages 2–13, 2004.

[4] T. Ball, S. Diehl, D. Notkin, and A. Zeller, editors. Number 05261 in Dagstuhl Seminar Proceedings. 2006.

[5] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

[6] L. Belady and M. Lehman. A Model of Large Program Development. *IBM Systems Journal*, 15(3):225–252, 1976.

[7] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. *ICSM*, 96:359–368.

[8] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA*, pages 166–177, 2000.

[9] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, pages 404–428, 2006.

[10] D. Dig and R. Johnson. How do apis evolve? a story of refactoring. *Journal of Software Maintenance*, 18(2):83–107, 2006.

[11] B. Fluri and H. Gall. Classifying Change Types for Qualifying Change Couplings. *ICPC*, pages 14–16, 2006.

[12] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[13] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *ICSM*, pages 190–197, 1998.

[14] C. Görg and P. Weißgerber. Error detection by refactoring reconstruction. In *MSR '05*, pages 29–35.

[15] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *OOPSLA*, pages 312–326, 2001.

[16] J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support API evolution. In *ICSE*, pages 274–283, 2005.

[17] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI*, volume 25, pages 234–245, June 1990.

[18] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[19] D. Jackson and D. A. Ladd. Semantic Diff: A tool for summarizing the effects of modifications. In *ICSM*, pages 243–252, 1994.

[20] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE TSE*, 28(7):654–670, 2002.

[21] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *ICSM*, pages 736–743, 2001.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. *ECOOP*, pages 327–353, 2001.

[23] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *MSR*, pages 58–64, 2006.

[24] S. Kim, K. Pan, and J. E. James Whitehead. When functions change their names: Automatic detection of origin relationships. In *WCRE*, pages 143–152, 2005.

[25] G. Malpohl, J. J. Hunt, and W. F. Tichy. Renaming detection. *Autom. Softw. Eng.*, 10(2):183–202, 2000.

[26] D. Mandelin, D. Kimelman, and D. Yellin. A bayesian approach to diagram matching with application to architectural models. In *ICSE*, pages 222–231, 2006.

[27] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.

[28] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *MSR*, pages 2–6, 2005.

[29] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM TOSEM*, 6(2):173–210, 1997.

[30] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA '02*, pages 97–106, 2002.

[31] F. Van Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. *IWPSE*, pages 126–130, 2003.

[32] Z. Wang, K. Pierce, and S. McFarling. BMAT - a binary matching tool for stale profile propagation. *J. Instruction-Level Parallelism*, 2, 2000.

[33] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE*, pages 231–240, 2006.

[34] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. *ASE*, pages 54–65, 2005.

[35] W. Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21(7):739–755, 1991.

[36] A. T. T. Ying, G. C. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, 2004.

[37] X. Zhang and R. Gupta. Matching execution histories of program versions. In *ESEC/SIGSOFT FSE*, pages 197–206, 2005.

[38] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31(6):429–445, 2005.

[39] L. Zou and M. W. Godfrey. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, 2005.